- ORIGINAL ARTICLE -

H-RADIC: A Fault Tolerance Framework for Virtual Clusters on Multi-Cloud Environments

H-RADIC: Una Solución de Tolerancia a Fallos para Clústeres Virtuales en Ambientes Multi-Nube

Ambrosio Royo, Jorge Villamayor, Marcela Castro-León, Dolores Rexachs and Emilio Luque

CAOS – Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Bellaterra (Cerdanyola del Vallès), Barcelona 08193, Spain
pabloambrosio.royo@e-campus.uab.cat,
{jorgeluis.villamayor,marcela.castro,dolores.rexachs,emilio.luque}@uab.cat

Abstract

Even though the cloud platform promises to be reliable, several availability incidents prove that it is not. How can we be sure that a parallel application finishes it´s execution even if a site is affected by a failure? This paper presents H-RADIC, an approach based on RADIC architecture, that executes parallel applications protected by RADIC in at least 3 different virtual clusters or sites. The execution state of each site is saved periodically in another site and it is recovered in case of failure. The paper details the configuration of the architecture and the experiment´s results using 3 clusters running NAS parallel applications protected with DMTCP, a very well-known distributed multi-threaded checkpoint tool. Our experiments show that by adding a cluster protector it will be possible to implement the next level in the hierarchy, where the first level in the RADIC hierarchy works as an observer at a site level. In adition, the experiments showed that the protection implementation is out of the critical path of the application and it depends on the utilized resources.

Keywords: Cloud, Fault-Tolerance, High-Performance Computing, RADIC.

Resumen

Aunque las plataformas en la nube parecen ser muy confiables, varios incidentes de disponibilidad han demostrado que no son tan confiables. ¿Cómo podemos asegurarnos que una aplicación paralela termina su ejecución cuando el sitio en la nube ha sido afectado por una falla? Este articulo presenta H-RADIC, un enfoque basado en la arquitectura RADIC, esta ejecuta aplicaciones paralelas en al menos 3 diferentes sitios o clústeres virtuales, todos protegidos por RADIC, donde el estado de la ejecución de cada sitio es guardado periódicamente en otro de los sitios y de ahí es recuperado en el caso de una falla. El articulo detalla la configuración de la arquitectura y los resultados de los experimentos usando 3 clústeres ejecutando aplicaciones NAS en paralelo, protegidas con DMTCP (una herramienta para realizar múltiples checkpoints). Nuestros experimentos muestran que al agregar un protector del clúster es posible implementar un nivel más en la jerarquía de RADIC, donde el primer nivel funciona como observador. Los experimentos muestran que la implementación de este protector esta fuera del camino critico de la ampliación y depende solamente de la utilización de recursos.

Palabras Clave: Nube, Tolerancia a Fallos, Computación de Altas Prestaciones, RADIC.

1. Introduction

We know that there aren't any computers, big or small, safe from failures, we have seen big cloud providers fail, Windows Azure had availability problems for the Olympic Games in 2012 [1], Amazon Web Services has been affected by extreme weather [2] and by human error [3], also Google Cloud Platform was attacked by low-level software [4] and even Oracle Cloud's wide network error issue was fixed by restarting the network [5].

Since communication with a cloud can be lost due to a wide range of possible errors, this also causes

Citation: A. Royo, J. Villamayor, M. Castro-León, D. Rexachs and E. Luque. *H-RADIC: The Fault Tolerance Framework for Virtual Clusters on Multi-Cloud Environments.* Journal of Computer Science & Technology, vol. 18, no. 3, pp. 210-217, 2018

DOI: 10.24215/16666038.18.e24

Received: June 6, 2018 **Accepted:** October 31, 2018

loss of computational resources, power consumption and money; different authors have worked on preventing failures when there are parallel applications with message passing running on cloud environments. We have studied the work of Villamayor et. al [6] and Gomez et. al [7] in Fault-Tolerance (FT) and took advantage of the elasticity (easy provisioning of "hardware") of cloud computing.

We propose taking the Redundant Array of Distributed Independent Controllers (RADIC) architecture and hierarchically scaling it up to be applied to a fully automated, elastic FT framework for virtual clusters running on different cloud providers. This is capable of protecting applications running in private or public clouds from failures such as loss of virtual nodes during execution or loss of communication between clouds.

The next section presents some related state-of-the-art work and a description of the RADIC architecture. In section 3, we will describe an overall detail of the H-RADIC architecture, followed up by section 4, a summary of the experiments carried out from the implementation of the H-RADIC architecture, finishing with conclusions and future work in section 5.

2. Background

One of the fundamental aspects of FT is the cost in time, money and resources that the FT has, requiring a balance between resources and overhead during a fault free execution. In Japan, Bautista-Gomez et. al [8] proposed a low-overhead high-frequency multi-level checkpoint technique which implements a three level checkpoint scheme that compensates for the overhead of the FT by dedicating a thread of execution per node.

Another group from the USA, S. Di, Y. Robert, F. Vivien, F. Cappello et. al [9], set up an online two-level checkpoint model for HPC, where one level deals with logical problems such as transient memory errors and the other one deals with hardware crashes like node fails, related to our work, contributing an online solution that determines the optimal checkpoint patterns and doesn't require knowledge of the job length in advance.

Egwutuoaha et. al [10], from Australia, approached the problem by not relaying on spare node prior to a fail, aiming to reduce the time and cost of the execution in the cloud.

In Spain, the approach of Gomez et. al [7], proposed a multi-cloud FT framework that was capable to continue working if any of the cloud sites fail and delivered the results on the due date.

Finally, the work of Villamayor et. al [6] where they propose Resilience as a Service (RaaS), a FT framework for High Performance Computing (HPC) applications running in a cloud. RaaS is built on top of the RADIC architecture and provides clouds with highly available, distributed and scalable FT services, from where we focus the main idea of this paper.

2.1. RADIC Architecture

RADIC defines an architecture to tolerate failures in nodes for parallel applications of message passing, RADIC functionalities are transparent and automatic, therefore the application doesn't have to be modified to apply it and there is no need for human intervention, it is also elastic since it has the ability to add new nodes whenever one crashes [11]. It consists of implementing FT for message passing applications, by intercepting and masking errors which detect and manages them instead of ending the application. Taking advantage of the hardware redundancy of nodes, implicit in cluster environments, RADIC needs at least three physical nodes to work so the application doesn't have to stop and start again. It works with two distributed software RADIC components [12]: Observers and Protectors. there is a protector running in each node and one observer for every process running in the application, as shown in Figure 1. Additionally, a table named RadicTable is used to store the relation between nodes, observers and protectors which is updated by the RADIC Components.

Observer: When the application is launched, a software instance is created and attached to every processor used in the program, its job it to safeguard the work done by its processor and mask any failures. It also performs checkpoints and send it to the Protector to be stored.

Protector: It is in charge of requesting the observers to perform checkpoints and storing the checkpoint files in its own Stable Storage (SS). It is also in charge of the detection of failures by verifying that the node that it's protecting is working and, should if fail, it performs the restoration of the process that failed by launching the latest checkpoint.

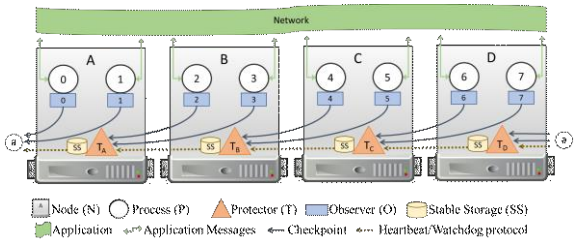

Figure 1 - RADIC Architecture

RADIC defines four functions: *Protection*, *Detection*, *Recovery* and *Error Masking*. The following functions will be described using as reference Figure 1.

Protection: Observer O_3 is in charge of monitor Process P_3, then O_3 gets a Protector T_A located in a different node N_A. At some point T_A will ask O_3 to get the state of it, process it and send it to T_A to be stored in T_A´s Stable Storage (SS). This is called a checkpoint. Checkpoints will be carried out periodically to keep on saving the state of the processes during a fault free execution. When every T has its neighbor's checkpoint, the RADIC protection will be in place.

Detection: Node N_B faults when the process P_3 cannot communicate with another process P_2 in the same node, which is when O_3 sends the error message to T_A. In addition, a node faults when there is no communication between nodes, that is, each protector (T_A) is in charge of detecting a possible failure in the neighbor Node (N_B). Each protector keeps a heartbeat/watchdog protocol with its neighbors Protectors, in this way a fault detection mechanism is implemented; on one hand, the Protector T_B is periodically sending heartbeats to T_C, and on the other hand, T_B is the watchdog of T_A, so if T_B loses communication with both neighbor protectors, T_B will destroy itself because it has been left alone, if T_A and T_C can't see T_B, then T_A will launch the recovery and error masking functions. Based on this function, RADIC needs at least 3 nodes to work properly.

Recovery: Protector T_A restarts/rolls-back the processes running in N_B, using the data saved in the SS from the last checkpoint, if the system has a spare node ($N_{B'}$), the processes will be restarted in it, otherwise the processes are restarted in the node (N_A) that has the checkpoint.

Error Masking: After the recovery function, when the processes of N_B have been restarted on $N_{B'}$, the protector T_A updates the RADIC Table and communications will be carried out as usual. The update of the new node in the system will be updated on demand, that is, if later the process P_1 tries to communicate with the old P_2 and cannot, the observer O_1 will tell the Protector T_A of the error in the communication, but since T_A knows that P_2 has been restarted somewhere else, T_A will send O_1 the new P_2's address.

RADIC has been originally designed to protect a parallel application with message passing, recently it has been leveraged to work in cloud environments. In our work we scale up the architecture to be used for parallel application protection, running in several virtual clusters on multiple cloud environments.

3. H-RADIC Architecture

We propose a new protection level of RADIC, the Hierarchical RADIC (H-RADIC) architecture, an automated and elastic FT framework that protects parallel applications with a Message Passing Interface (MPI) running in Virtual Clusters on Cloud (VCC); At least three VCC are required, each one of them protected with the RADIC architecture and located in different geographical sites so they don't share any of the physical resources. This will allow us to identify non-virtual nodes faults within and between the VCC.

Besides the regular RADIC architecture designed to protect from node fails, H-RADIC will protect applications from crashing in the event of multiple fails, granting that the application finishes its execution despite the fails. When the virtual nodes fail, the physical nodes that the virtual ones are mounted on fail and/or whenever there is loss of

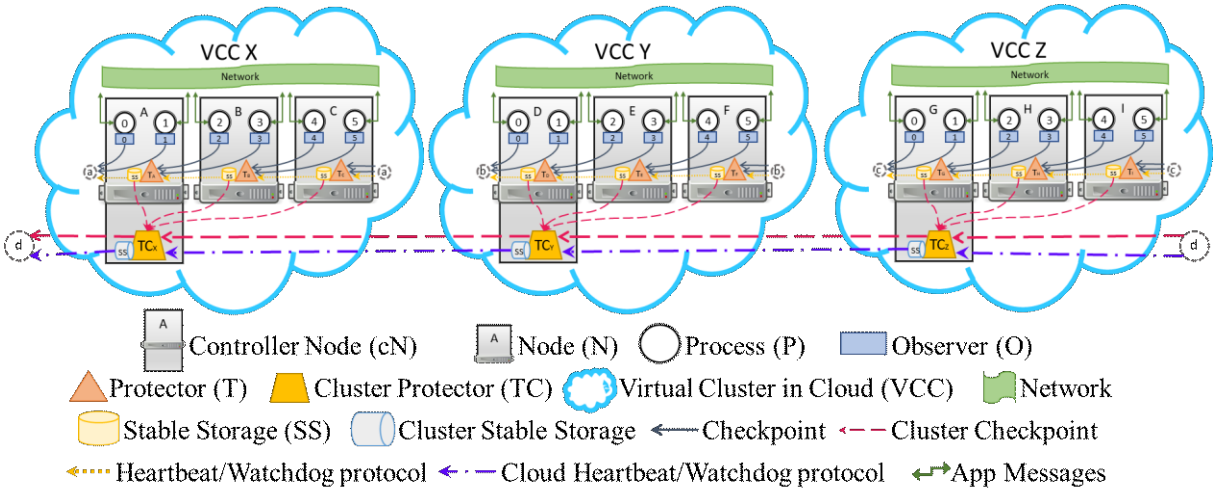

Figure 2 - H-RADIC Architecture

communication, H-RADIC will perform a diagnostic and then management of the error.

Additional to all the RADIC components, H-RADIC has an extra software component, the proTector of the Cluster (TC); on every VCC, a main node will be defined (controller Node - cN) to create the TC´s, which will be in charge of the communications between the VCCs, as its shown in Figure 2.

3.1. H-RADIC functions

The H-RADIC architecture implements the RADIC functions in each cluster, that is, whenever there is a fault in the cluster, the fault tolerant system tries to recover from it by applying the RADIC functions. H-RADIC functions will process the failures when the RADIC fault tolerance in the cluster cannot process them or whenever there is a loss of communication with a site.

To guarantee the completion of the execution, H-RADIC transports the RADIC functions to be applied into programs running in multi-cluster environments and depict its functionality in the following functions:

Protection: Only while the execution is running fault free, checkpoints will be periodically carried out. Once the Protectors (T) have the checkpoint of each process, the Cluster Protector (TC_Y) will be in charge of collecting the updates in the Stable Storage (SS) of every T in the Cluster (VCC_Y). Then, TC_Y must send it to the assigned Cluster Protector TC_X located in a different cloud to store the Cluster Checkpoint (multilevel checkpoint) at the Cluster SS. When every TC has a checkpoint of its neighbor VCC, the H-RADIC protection will be activated.

Detection: This function works the same way as the RADIC detection function; TC_X is in charge of identifying a possible failure in the neighbor Cluster VCC_Y using a heartbeat/watchdog protocol. If VCC_Y does not answer, TC_X asks TC_Z to verify if there is an error with VCC_Y, if TC_Z confirms that there is no communication with VCC_Y, Recovery and Error Masking functions are triggered.

Recovery: Cloud Protector TC_X restarts/rolls-back the process running in VCC_Y, using the data saved in the cluster SS from the last checkpoint, then TC_X checks if there is a spare cluster to launch the checkpoint in it, otherwise new nodes are created in VCC_X to restart all the processes previously running in the failed VCC_Y, then the RADIC Tables are updated.

Error Masking: After the recovery function, when the processes have been restarted, the Cluster Protector hides the communication errors caused by the cloud failure, the Cluster Protector TC_X sends a message to the affected Protectors with the new address where the processes have been recovered, updating the H-RADIC Table, then communications will be carried out as usual. The update of the new cluster in the system will be updated on demand.

3.1.1. H-RADIC Recovery function' options

After a fault is detected, the recovery function has two options to restart a checkpoint:

1) Check if there are spare clusters available in another cloud, viewed in Figure 3.
2) Check if there are spare nodes in the same cloud, as shown in Figure 4.

Then the checkpoint files are sent to the nodes that are in the spare cluster.

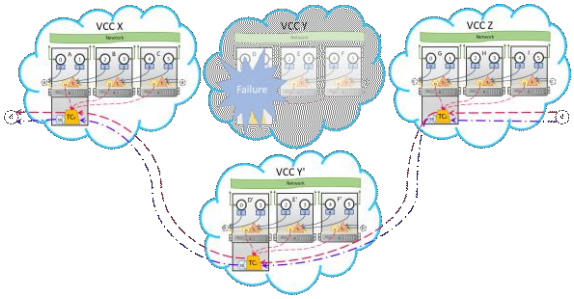

Figure 3 - H-RADIC Recovery function - spare nodes/cluster in another cloud.

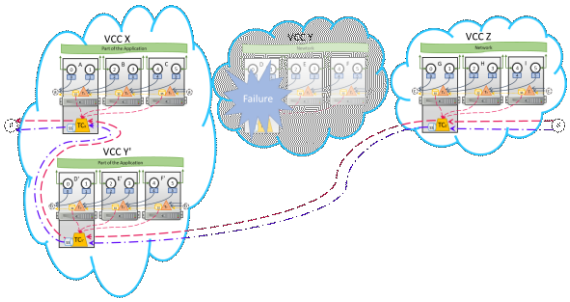

Figure 4 - H-RADIC Recovery function - spare nodes/cluster in the same cloud.

The main difference between the two restart options is the way to store the checkpoints. In the case that there is a spare cluster available in another cloud (Figure 3), H-RADIC will be working as usual, but if there are no spare nodes available in a third cloud, the checkpoint will be restarted in the spare nodes of the cloud that has the checkpoint. After the restart, the two clusters (X and Y' in Figure 4) will send their checkpoints to Cluster Z and vice versa.

When the execution is working as in the previous situation (Figure 4) and if a new failure rises in one of the clouds, the three clusters will be together in one cloud, as shown in Figure 5, and henceforth, all the checkpoints will be stored in one cloud storage.

Although this situation will leave the execution exposed to the same vulnerabilities, this leads us to develop this type architecture. There are vulnerabilities such as failures in the cloud

controller, the storage and the physical computer that the virtual nodes are mounted on. These vulnerabilities will now allow us to be able to guaranty availability.

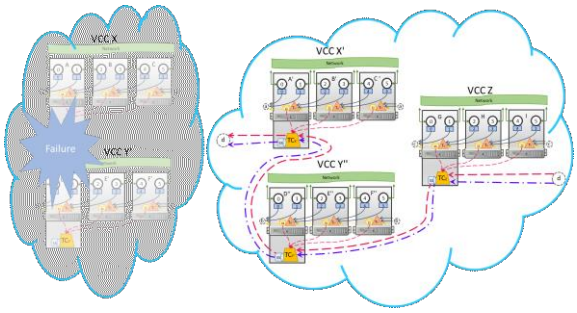

Figure 5 - H-RADIC Recovery function - spare nodes/cluster in the same cloud and not more clouds left.

3.2. Pseudocode of H-RADIC

Before running the programs, they have to be compiled with the MPI implementation and also the inventory of nodes has to be depicted in the H-RADIC Table (Table *1*); it's been established that each cluster will have a Cluster Protector, which will be the previous row in the table, and it's expressed as (cluster-1), if the program is in VCC Y, the cluster protector will be VCC X. If the cluster it's in the first row (VCC X), then the cluster protector will be the last row in the table (VCC Z), to complete a full loop. The spare clusters will be added to the table and whenever the application needs to use a spare cluster, it looks for it in the RADIC Table.

All the logic described in the H-RADIC functions can be summarized with the following pseudocode. Algorithm 1 describes the Protection function.

In Algorithm 2 we can find the pseudocode description of the detection function.

The recovery function pseudocode is described in Algorithm 3, this will be called only if the detection function found an error there.

Finally, the error masking function pseudocode is described in Algorithm 4.

Table 1 - H-RADIC TABLE

	Cluster	*Nodes*	*Controller Node*	*Cluster Protector*
1	VCC X	A,B,C	A	Z
2	VCC Y	D,E,F	E	X
3	VCC Z	G,H,I	G	Y
4	spareClust	J,K,L	J	-

Algorithm 1 - H-RADIC Protection function pseudocode

```
1:  function protection (program, cluster,
    ckPTime){
2:    Send the program to the controllerNode in
    cluster
3:      launch dmtcpCoordinator with the program
    and ckPTime
4:      call detection(cluster, ckPTime)
5:      while
6:        perform checkPoint every ckPTime
    seconds
7:          send checkPoint to (cluster-1)
8:        if program ends
9:          successful end
10:         break while
11: }
```

Algorithm 2 - H-RADIC Detection function pseudocode

```
1:  function detection (cluster, ckPTime){
2:    // Goes to the HRADIC Table and takes the
    next cluster in the table to protect
3:    establish heartbeat/watchdog protocol
    between cluster and (cluster+1)
4:    while
5:      send heartbeat to (cluster-1) every certain
    time
6:      if heartbeat/watchdogProtocol fails
7:        ask (cluster+2) to establish connection
    with (cluster+1)
8:        if the connection is established
9:          nothing happens
10:       else if (cluster+2) can't see (cluster+1)
11:         recovery(cluster+1, ckPTime)
12: }
```

Algorithm 3 - H-RADIC Recovery function pseudocode

```
1:  function recovery (cluster+1, ckPTime){
2:    search for spareCluster in other clouds or in
    the local cloud
3:    inizialize Nodes in spareCluster = Nodes in
    (cluster+1)
4:    send checkPoint files to nodes in
    spareCluster
5:    call errorMasking(spareCluster, cluster+1)
6:    // Call the protection fuction again, but this
    time run the checkpoint instead of the original
    program
7:    call
    protection(checkPoint,spareCluster,ckPTime)
8:  }
```

Algorithm 4 - H-RADIC Error Masking function pseudocode

```
1: function errorMasking (spareCluster, cluster+1){
2:   remove (cluster+1) info. in the HRADIC Table
3:   move spareCluster info. in the HRADIC Table to where (cluster+1) was.
4: }
```

All of these Algorithms are called subsequently, when the application is launched the protection function starts as well; the protection calls the detection function and if the detection function identifies an error, it calls the recovery and error masking functions.

4. Experiments

To test the architecture, we mounted 3 clusters running on CentOS. Each one of them has a different NAS Parallel Benchmark [13] program compiled with an implementation of MPI named: MPICH [14]. H-RADIC will performing the checkpoints for these experiments in a coordinated approach, using the open source software package: Distributed Multi-Threaded CheckPointing (DMTCP) [15]. Each cluster has 3 nodes and every node have 8 cores, 24 cores per cluster.

The DMTCP allow us to work with the checkpoints at an application layer of the OSI model, making it perfect for virtualized environments. Moreover, whenever a checkpoint is done, it automatically compresses it using gzip.

4.1. Results

In this section, we exemplify the different resources that RADIC needs for its initial configuration, such as the size of the checkpoints and the time to perform checkpoints. To evaluate the cost of storage and time during the execution, we measure the time that the application takes to perform a checkpoint and the size of it, and the time of RADIC and H-RADIC when they activate.

Calculating the optimal checkpoint interval is a controversial subject; for these experiments, all the programs were executed several times, which was done to identify the average time that each node needed to perform a checkpoint and move it to the different nodes and also to the cluster protector.

The experiment is measuring the time that it takes the application to finish its execution, in the following cases: 1) the application without applying H-RADIC but performing checkpoints, 2) the application with H-RADIC and without errors or failures and, 3) the application with H-RADIC and an induced error. Then by taking the increase percentage of time in 2) and 3), we attain Figure 6.

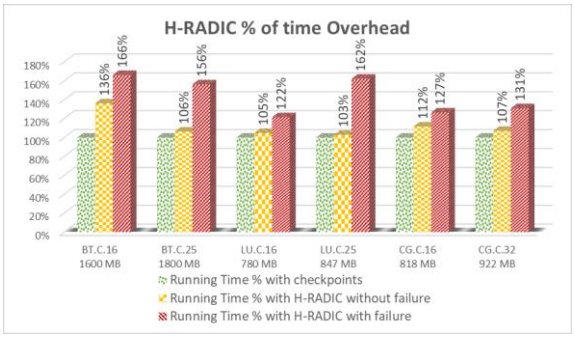

Figure 6 - H-RADIC percentage of time overhead.

In general, we can appreciate that the overhead caused by implementing H-RADIC and having 0 failures, does not really increase the program execution time. However, when we induce the error, we can see the overhead percentage from as low as 22% to as high as 66% in time. This bottleneck is mainly due to the time taken to move the checkpoint to the nodes in the spare cluster, since the time to restart the application once the checkpoints are in the spare cluster is negligible.

Another experiment was performed where an error was induced around the middle of the application´s execution and the we took the time that the application took to restart form the latest checkpoint and the time that the application took to restart from the beginning, as shown in Figure 7.

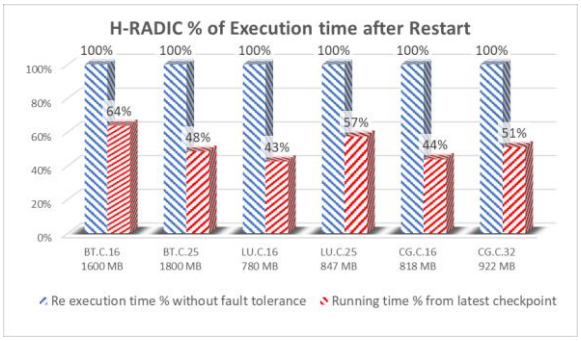

Figure 7 - Execution time after restart

4.1.1. Overhead Breakdown

For the calculation of the overhead, we measured different time variables (as shown in Table 2): the first percentage shown in Figure 6 is the application running with checkpoints as Equation(1). This time is considered the point of departure from where we are calculating the overhead.

$$T_{ex-ckpt} = T_{ex} + n\,x\,T_{ckpt} \quad \text{(Equation 1)}$$

In order to establish the RADIC and H-RADIC protection, all the checkpoints needed to be copied to the protector nodes and the cluster protector node

respectively. This process is done in the background Equation (2) while the application keeps on executing. This process is represented in Figure 2, by the checkpoint's and cluster checkpoint's lines. Since moving the files take some computational resources, it affects the program execution time, although it's not directly in the critical path.

$$\boldsymbol{overhead = ckptSize\ x\ T_{mv-ckpt}} \text{ (Equation 2)}$$

The *Running Time percentage with H-RADIC without failure* in Figure 6 is the time that the application took to execute considering the RADIC and H-RADIC protection in Equation (3).

$$\boldsymbol{T_{H-RADIC} = T_{ex-ckpt} + overhead} \text{ (Equation 3)}$$

Finally, the third experiment, the *Running Time percentage with H-RADIC with failure* in Figure 6 is the time that took it took the framework to restart the execution in a spare cluster Equation (4), considering the time to move the checkpoint files and the time to restart the execution.

$$\boldsymbol{T_{H-RADIC\ \&\ Failure}}$$

$$= T_{H-RADIC} + T_{mv-ckpt-spare} + T_{restart} \text{ (Equation 4)}$$

Table 2 - Variables description.

Variable	Description
T_{ex}	Program execution time
$T_{ex\text{-}ckpt}$	Execution time with checkpoints
T_{ckpt}	Time to create a checkpoint
n	Number of checkpoints per execution
$T_{H\text{-}RADIC}$	Execution time with H-RADIC without failures
ckptSize	Checkpoint size in MB
$T_{mv\text{-}ckpt}$	Time to move the ckpt to establish H-RADIC protection.
bkgrd	Unpredictable time impact on *p*
$T_{H\text{-}RADIC\ \&Failure}$	Execution time with H-RADIC with failure
$T_{mv\text{-}ckpt\text{-}spare}$	Time to move the checkpoint files to the spare cluster
$T_{restart}$	Time to restart the application

5. Conclusion and future work

The proposal in this paper is based on the RADIC architecture, which is capable of assuring a successful execution of the application even when failures occur. We analyzed every component of the framework and identified the improvement areas.

By translating every RADIC concept to H-RADIC and taking them to a virtual multi-cluster level, we develop a FT framework capable of overcoming fatal fails like the loss of a full site. The H-RADIC architecture fully implements RADIC and the new benefits from H-RADIC and it allows us to guarantee the completion of the execution in the event of errors such as: loss of nodes, loss of communication between sites or general fails in several sites. It's a solution that can be implemented in virtual and non-virtual environments.

Considering that a traditional FT system, which consist on having one or two copies of a full site in another one, waiting for the main site to fail, with H-RADIC, by distributing the work load in different sites, the need of resources is way less that the ones needed for the traditional FT system.

Finally, the experiments show us that the overhead in most of the programs is reasonable and proves the theory behind the concept.

The architecture can be tested by implementing semi coordinated and no-coordinated checkpoints, in order to run a single program in several virtual clusters at the same time.

Develop the architecture in a way that if the performance is decreasing, it can request more resources live to the cloud provider, to ensure completion before a deadline.

Implement a mechanism that accelerates the transfer rate, by improving I/O, use incremental checkpoints and/or compress [16] the checkpoints even more.

Acknowledgements

The first author acknowledges the support from the National Science and Technology Council of Mexico (Consejo Nacional de Ciencia y Tecnología, CONACYT) that sponsored through a scholarship, and special thanks to all the team at CAOS and the UAB. This research has been supported by the Agencia Estatal de Investigación (AEI), Spain and the Fondo Europeo de Desarrollo Regional (FEDER) UE, under contract TIN2017-84875-P and partially funded by a research collaboration agreement with the Fundacion Escuelas Universitarias Gimbernat (EUG).

Competing interests

The authors have declared that no competing interests exist.

References

[1] B. Darrow, "Windows Azure outage hits Europe," 26-Jul-2012. [Online]. Available: https://gigaom.com/2012/07/26/windows-azure-outage-hits-europe/. [Accessed: 30-Mar-2018].

[2] O. Malik, "Severe storms cause Amazon Web Services outage," 29-Jun-2012. [Online]. Available:

https://gigaom.com/2012/06/29/some-of-amazon-web-services-are-down-again/. [Accessed: 30-Mar-2018].

[3] "Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/message/41926/. [Accessed: 31-Mar-2018].

[4] "Google Cloud Status Dashboard." [Online]. Available: https://status.cloud.google.com/incident/storage/17002. [Accessed: 31-Mar-2018].

[5] J. Hult, "Oracle Cloud - unplanned outage - November 7, 2017," *JonathanHult.com*, 17-Nov-2017. .

[6] J. Villamayor, D. Rexachs, and E. Luque, "RaaS: Resiliance as a Service – Fault Tolerance for High Performance Computing in Clouds," presented at the International Symposium on Cluster, Cloud and Grid Computing, 2018, p. Accepted.

[7] A. Gómez, L. M. Carril, R. Valin, J. C. Mouriño, and C. Cotelo, "Fault-tolerant virtual cluster experiments on federated sites using BonFIRE," *Future Gener. Comput. Syst.*, vol. 34, pp. 17–25, May 2014.

[8] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2011, p. 32:1–32:32.

[9] S. Di, Y. Robert, F. Vivien, and F. Cappello, "Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 244–259, Jan. 2017.

[10] I. P. Egwutuoha, S. Chen, D. Levy, B. Selic, and R. Calvo, "Cost-oriented proactive fault tolerance approach to high performance computing (HPC) in the cloud," *Int. J. Parallel Emergent Distrib. Syst.*, vol. 29, no. 4, pp. 363–378, Jul. 2014.

[11] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. Luque, "Challenges and Issues of the Integration of RADIC into Open MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, Berlin, Heidelberg, 2009, pp. 73–83.

[12] M. Castro-León, H. Meyer, D. Rexachs, and E. Luque, "Fault tolerance at system level based on RADIC architecture," *J. Parallel Distrib. Comput.*, vol. 86, pp. 98–111, Dec. 2015.

[13] "NAS Parallel Benchmarks," *NASA Advanced Supercomputing Division*. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html. [Accessed: 23-May-2018].

[14] "MPICH | High-Performance Portable MPI," *MPICH*. [Online]. Available: https://www.mpich.org/. [Accessed: 02-Jun-2018].

[15] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[16] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Improving Performance of Iterative Methods by Lossy Checkponting," *ArXiv180411268 Cs*, Apr. 2018.